Approximate matching

Ben Langmead

You are free to use these slides. If you do, please sign the guestbook (www.langmead-lab.org/teaching-materials), or email me (ben.langmead@gmail.com) and tell me briefly how you’re using them. For original Keynote files, email me.
Read alignment requires approximate matching

Sequence differences occur because of...

1. Sequencing error
2. Natural variation
Approximate string matching

Looking for places where a $P$ matches $T$ with up to a certain number of mismatches or edits. Each such place is an approximate match.

A mismatch is a single-character substitution:

$$
\begin{align*}
T: & \quad GGAAAAAGAGGTAGCGGCTTTAACAGTAG \\
& \quad | \quad | \quad | \quad | \quad | \quad | \\
P: & \quad GTAACGGCG
\end{align*}
$$

An edit is a single-character substitution or gap (insertion or deletion):

$$
\begin{align*}
T: & \quad GGAAAAAGAGGTA\textcolor{red}{GC}GGCTTTAACAGTAG \\
& \quad | \quad | \quad | \quad | \quad | \quad | \\
P: & \quad GTA\textcolor{red}{AC}GGCG
\end{align*}
$$

$$
\begin{align*}
T: & \quad GGAAAAAGAGGTA\textcolor{red}{AGC} - \textcolor{green}{GCG}TTTTAACAGTAG \\
& \quad | \quad | \quad | \quad | \quad | \quad | \\
P: & \quad GTA\textcolor{green}{AGCGGCG}
\end{align*}
$$

$$
\begin{align*}
T: & \quad GGAAAAAGAGG\textcolor{red}{TAG}CGGCTTTAACAGTAG \\
& \quad | \quad | \quad | \quad | \quad | \quad | \\
P: & \quad GT\textcolor{red}{-GC}GCGCG
\end{align*}
$$
Hamming and edit distance

For two same-length strings $X$ and $Y$, *hamming distance* is the minimum number of single-character substitutions needed to turn one into the other:

$X$: GAGGTAGCGGCGTTTAAC

$Y$: GTGGTAACGGGGTTTAAC

Hamming distance = 3

*Edit distance (Levenshtein distance)*: minimum number of *edits* required to turn one into the other:

$X$: TGCCGCGCAAAAACAGC

$Y$: TGACCGCGCAAAACAGC

Edit distance = 2

$X$: GCTATGCGGCTAACGC

$Y$: GCTATGCGGCTATACGC

Edit distance = 2

$X$: GC - TATGCGGCTATACGC

$Y$: GC - TATGCGGCTATACGC

Edit distance = 2
Approximate string matching

Adapting the naive algorithm to do approximate string matching within configurable Hamming distance:

```python
def naiveApproximate(p, t, maxHammingDistance=1):
    occurrences = []
    for i in xrange(0, len(t) - len(p) + 1):  # for all alignments
        nmm = 0
        for j in xrange(0, len(p)):
            # for all characters
            if t[i+j] != p[j]:  # does it match?
                nmm += 1  # mismatch
                if nmm > maxHammingDistance:
                    break  # exceeded maximum distance
        if nmm <= maxHammingDistance:
            # approximate match; return pair where first element is the
            # offset of the match and second is the Hamming distance
            occurrences.append((i, nmm))
    return occurrences
```

Instead of stopping upon first mismatch, stop when maximum distance is exceeded

Approximate string matching

How to make Boyer-Moore and index-assisted exact matching approximate?

Helpful fact: Split $P$ into non-empty non-overlapping substrings $u$ and $v$. If $P$ occurs in $T$ with 1 edit, either $u$ or $v$ must match exactly.

More generally: Let $p_1, p_2, ..., p_{k+1}$ be a partitioning of $P$ into $k+1$ non-overlapping non-empty substrings. If $P$ occurs in $T$ with up to $k$ edits, then at least one of $p_1, p_2, ..., p_{k+1}$ must match exactly.

≤ $k$ edits can affect as many as $k$ of these, but not all
Approximate string matching

These rules provide a bridge from the exact-matching methods we’ve studied so far, and approximate string matching.

\[ p_1 \quad p_2 \quad p_3 \quad p_4 \quad \ldots \quad p_{k+1} \]

\( \leq k \) edits can overlap as many as \( k \) of these, but not all

Use an exact matching algorithm to find exact matches for \( p_1, p_2, \ldots, p_{k+1} \). Look for a longer approximate match in the vicinity of the exact match.

Use an exact matching algorithm to find exact matches for \( p_1, p_2, \ldots, p_{k+1} \). Look for a longer approximate match in the vicinity of the exact match.
Approximate string matching

def bmApproximate(p, t, k, alph="ACGT"):
    """ Use the pigeonhole principle together with Boyer-Moore to find
    approximate matches with up to a specified number of mismatches. """
    if len(p) < k+1:
        raise RuntimeError("Pattern too short (%d) for given k (%d)" % (len(p), k))
    ps = partition(p, k+1)  # split p into list of k+1 non-empty, non-overlapping substrings
    off = 0  # offset into p of current partition
    occurrences = set()  # note we might see the same occurrence >1 time
    for pi in ps:  # for each partition
        bm_prep = BMPreprocessing(pi, alph=alph)  # BM preprocess the partition
        for hit in bm_prep.match(t)[0]:
            if hit - off < 0: continue  # pattern falls off left end of T?
            if hit + len(p) - off > len(t): continue  # falls off right end?
            # Count mismatches to left and right of the matching partition
            nmm = 0
            for i in range(0, off) + range(off+1, len(p)):
                if t[hit-off+i] != p[i]:
                    nmm += 1
                    if nmm > k: break  # exceeded maximum # mismatches
            if nmm <= k:
                occurrences.add(hit-off)  # approximate match
    off += len(pi)  # Update offset of current partition
    return sorted(list(occurrences))

Approximate Boyer-Moore performance

<table>
<thead>
<tr>
<th></th>
<th>Boyer-Moore, exact</th>
<th>Boyer-Moore, ≤1 mismatch with pigeonhole</th>
<th>Boyer-Moore, ≤2 mismatches with pigeonhole</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># character</td>
<td>wall clock</td>
<td># matches</td>
</tr>
<tr>
<td>comparisons</td>
<td>comparisons</td>
<td>time</td>
<td></td>
</tr>
<tr>
<td><strong>P:</strong> “tomorrow”</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>T:</strong> Shakespeare’s complete works</td>
<td>786 K</td>
<td>1.91s</td>
<td>17</td>
</tr>
<tr>
<td><strong>P:</strong> 50 nt string from Alu repeat*</td>
<td>32.5 M</td>
<td>67.21 s</td>
<td>336</td>
</tr>
<tr>
<td><strong>T:</strong> Human reference (hg19) chromosome 1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* GCGCGGTGGCTCAGCCTGTAATCCCAAGCACCTTGGGAGGCGAGGCGGGG

P: “tomorrow”
T: Shakespeare’s complete works
P: 50 nt string from Alu repeat*
T: Human reference (hg19) chromosome 1

Johns Hopkins Whiting School of Engineering
Approximate string matching: more principles

Let $p_1, p_2, ..., p_{k+1}$ be a partitioning of $P$ into $k+1$ non-overlapping non-empty substrings. If $P$ occurs in $T$ with up to $k$ edits, then at least one of $p_1, p_2, ..., p_{k+1}$ must match exactly.

Let $p_1, p_2, ..., p_j$ be a partitioning of $P$ into $j$ non-overlapping non-empty substrings. If $P$ occurs with up to $k$ edits, then at least one of $p_1, p_2, ..., p_j$ must occur with $\leq \text{floor}(k / j)$ edits.
### Review: approximate matching principles

#### Non-overlapping substrings

<table>
<thead>
<tr>
<th>Pigeonhole principle</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p_1, p_2, \ldots, p_i ) is a partitioning of ( P ). If ( P ) occurs with ( \leq k ) edits, at least one partition matches with ( \leq \text{floor}(k / j) ) edits.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pigeonhole principle with ( j = k + 1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p_1, p_2, \ldots, p_{k+1} ) is a partitioning of ( P ). If ( P ) occurs in ( T ) with ( \leq k ) edits, at least one partition matches exactly.</td>
</tr>
</tbody>
</table>

#### General

Let \( j = k + 1 \)

- **Why?**
  - Smallest value s.t. \( \text{floor}(k / j) = 0 \)

- **Why make \( \text{floor}(k / j) = 0 \)?**
  - So we can use exact matching

#### Specific

- **Why is smaller \( j \) good?**
  - Yields fewer, longer partitions

- **Why are long partitions good?**
  - Makes exact-matching filter more specific, minimizing # candidates
Approximate string matching: more principles

We partitioned $P$ into non-overlapping substrings

Consider overlapping substrings
Approximate string matching: more principles

Say substrings are length \( q \). There are \( n - q + 1 \) such substrings.

Worst case: 1 edit to \( P \) changes up to \( q \) substrings

Minimum # of length-\( q \) substrings unedited after \( k \) edits? \( n - q + 1 - kq \)

\( q \)-gram lemma: if \( P \) occurs in \( T \) with up to \( k \) edits, alignment must contain \( t \) exact matches of length \( q \), where \( t \geq n - q + 1 - kq \)
Approximate string matching: more principles

If $P$ occurs in $T$ with up to $k$ edits, alignment contains an exact match of length $q$, where $q \geq \text{floor}(n / (k + 1))$

Derived by solving this for $q$: $n - q + 1 - kq \geq 1$

Exact matching filter: find matches of length $\text{floor}(n / (k + 1))$ between $T$ and any substring of $P$. Check vicinity for full match.
## Approximate matching principles

<table>
<thead>
<tr>
<th>Non-overlapping substrings</th>
<th>Overlapping substrings</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pigeonhole principle</strong></td>
<td><strong>q-gram lemma</strong></td>
</tr>
<tr>
<td>$p_1, p_2, ..., p_i$ is a partitioning of $P$. If $P$ occurs with $\leq k$ edits, at least one partition matches with $\leq \text{floor}(k / j)$ edits.</td>
<td>If $P$ occurs with $\leq k$ edits, alignment contains $t$ exact matches of length $q$, where $t \geq n - q + 1 - kq$</td>
</tr>
<tr>
<td><strong>Pigeonhole principle with $j = k + 1$</strong></td>
<td><strong>q-gram lemma with $t = 1$</strong></td>
</tr>
<tr>
<td>$p_1, p_2, ..., p_{k+1}$ is a partitioning of $P$. If $P$ occurs in $T$ with $\leq k$ edits, at least one partition matches exactly.</td>
<td>If $P$ occurs with $\leq k$ edits, alignment contains an exact match of length $q$ where $q \geq \text{floor}(n / (k + 1))$</td>
</tr>
</tbody>
</table>
Sensitivity

Sensitivity = fraction of “true” approximate matches discovered by the algorithm

*Lossless* algorithm finds all of them, *lossy* algorithm doesn’t necessarily

We’ve seen *lossless* algorithms. Most everyday tools are *lossy*. Lossy algorithms are often much speedier & still acceptably sensitive (e.g. BLAST, BLAT, Bowtie).

Example lossy algorithm: pick $q > \text{floor}(n / (k + 1))$